

EV355228467

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

Methods and Systems for Transparent Depth Sorting

Inventor(s):
Jeff Andrews

ATTORNEY'S DOCKET NO. ms1-1378us

TECHNICAL FIELD

This invention relates to the field of computer graphics. More specifically, the present invention pertains to methods and systems for rendering objects.

BACKGROUND

Computer graphics systems typically utilize instructions, implemented via a graphics program on a computer system, to specify calculations and operations needed to produce two-dimensional or three-dimensional displays. Exemplary graphics systems that include APIs that are commercially available for rendering three-dimensional graphs include Direct3D, available from Microsoft Corporation, of Redmond, Wash., and OpenGL by Silicon Graphics, Inc., of Mountain View, Calif.

Computer graphics systems can be envisioned as a pipeline through which data pass, where the data are used to define an image that is to be produced and displayed. At various points along the pipeline, various calculations and operations that are specified by a graphics designer are used to operate upon and modify the data.

In the initial stages of the pipeline, the desired image is described by the application using geometric shapes such as lines and polygons, referred to in the art as “primitives.” The derivation of the vertices for an image and the manipulation of the vertices to provide animation entail performing numerous geometric calculations in order to project the three-dimensional world being designed to a position in the two-dimensional world of the display screen.

Primitives are constructed out of “fragments.” These fragments have attributes calculated, such as color and depth. In order to enhance the quality of

1 the image, effects such as lighting, fog, and shading are added, and anti-aliasing
2 and blending functions are used to give the image a smoother and more realistic
3 appearance. The processes pertaining to per fragment calculation of colors, depth,
4 texturing, lighting, etc., are collectively known as “rasterization”.

5 The fragments and their associated attributes are stored in a frame buffer.
6 Once rasterization of the entire frame has been completed, pixel values can then be
7 read from the frame buffer and used to draw images on the computer screen.

8 To assist in understanding a typical computer graphics system, consider
9 Fig. 1 which illustrates, generally at 100, a system that can implement a computer
10 graphics process. System 100 comprises a graphics front end 102, a geometry
11 engine 104, a rasterization engine 106, and a frame buffer 108. System 100 can
12 typically be implemented in hardware, software, firmware or combinations
13 thereof, and is also referred to as a “rendering pipeline”.

14 Graphics front end 102 comprises, in this example, an application,
15 primitive data generation stage 102a and display list generation stage 102b. The
16 graphics front end generates primitive data consumed by the subsequent pipeline
17 stage(s). Primitive data is typically loaded from a computer system’s memory and
18 saved in a display list in the display list stage 102b. All geometric primitives are
19 eventually described by vertices or points.

20 Geometry engine 104 comprises, in this example, high order surface (HOS)
21 tessellation 104a, and per-vertex operations stage 104b. In stage 104a, primitive
22 data is converted into simple rasterizer-supported primitives (typically triangles)
23 that represent the surfaces that are to be graphically displayed. Some vertex data
24 (for example, spatial coordinates) are transformed by four-by-four floating point
25 matrices to project the spatial coordinates from a position in the three-dimensional

1 world to a position on the display screen. In addition, certain other advanced
2 features can also be performed by this stage. Texturing coordinates may be
3 generated and transformed. For example, lighting calculations can be performed
4 using the vertex, the surface normal, material properties, and other light
5 information to produce a color value. Perspective division, which is used to make
6 distant objects appear smaller than closer objects in the display, can also occur in
7 per-vertex operations stage 104b.

8 Rasterization engine 106 is configured to perform so-called rasterization of
9 the re-assembled rasterizer-supported primitives. It comprises the following
10 stages: triangle/point assembly 106a, setup 106b, parametric evaluation 106c,
11 depth and stencil operations stage 106d, per-fragment operations stage 106e, and
12 the blend and raster operations (ROP) stage 106f.

13 Rasterization refers to the conversion of vertex data connected as rasterizer-
14 supported primitives into “fragments.” Each fragment corresponds to a single
15 element (e.g., a “pixel” or “sub-pixel”) in the graphics display, and typically
16 includes data defining color, shading, depth, and texture. Thus, for a single
17 fragment, there are typically multiple pieces of data defining that fragment. To
18 perform its functions, triangle/point assembly stage 106a fetches different vertex
19 components, such as a texture component, a specular component, a fog
20 component, and an alpha blending component.

21 Setup stage 106b converts the vertex data into parametric function
22 coefficients that can then be evaluated on a fragment coordinate (either pixel or
23 sub-pixel) by fragment coordinate basis. Parametric evaluation stage 106c
24 evaluates the parametric functions for all the fragments which lie within the given
25 rasterizable primitive, while conforming to rasterizable primitive inclusion rules.

1 Depth and stencil operations stage 106d perform depth operations on the
2 projected fragment depth and application specified fragment stencil operations.
3 These operations apply to both the comparison function on the depth and stencil
4 values, how the depth and stencil values should be updated in the depth/stencil
5 buffer and whether the fragment should terminate or continue processing. In the
6 idealized rasterization pipeline these operations take place just before frame buffer
7 write-back (after blend and ROP stage 106f), but commonly these operations are
8 valid before the per-fragment operations stage 106e.

9 Per-fragment operations stage 106e typically performs additional
10 operations that may be enabled to enhance the detail or lighting effects of the
11 fragments, such as texturing, bump mapping, per-fragment lighting, fogging, and
12 other like operations. Near the end of the rasterization pipeline is the blend and
13 raster operation (ROP) stage 106f, which implements blending for transparency
14 effects and traditional 2D blit raster operations. After completion of these
15 operations, the processing of the fragment is complete and it is typically written to
16 frame buffer 110 and potentially to the depth/stencil buffer 108. Thus, there are
17 typically multiple pieces of data defining each pixel.

18 Now consider so-called “depth sorting” as it pertains to rendering 3-D
19 graphics. Depth sorting in computer graphics is typically accomplished using
20 what is referred to as a “depth buffer”. A depth buffer, often called a “z-buffer” or
21 a “w-buffer”, is a property of a graphics device that stores depth information to be
22 used by the graphics device to render a scene. Typically, when a graphics device
23 renders a 3-D scene to a target surface, it can use the memory in an associated
24 depth-buffer surface as a workspace to determine how the pixels or sub-pixels of
25 rasterized polygons occlude one another. The target surface typically comprises

1 the surface or buffer to which final color values are written. The depth-buffer
2 surface that is associated with the render-target surface is used to store depth
3 information that tells the graphics device how deep each visible pixel or sub-pixel
4 is in the scene.

5 When a 3-D scene is rasterized in a rasterization pipeline with depth
6 buffering enabled, each point on the rendering surface is tested. The values in the
7 depth buffer can be a point's z-coordinate or its homogeneous w-coordinate—
8 from the point's (x, y, z, w) location in projection space. A depth buffer that uses z
9 values is often called a z-buffer, and one that uses w values is called a w-buffer.

10 At the beginning of the test, the depth value in the depth buffer is set to the
11 largest possible value for the scene. The color value on the rendering surface is set
12 to either the background color value or the color value of the background texture
13 at that point. Each polygon in the scene is tested to see if it intersects with the
14 current coordinate (x,y) on the rendering surface. If it does, the depth value—
15 which will be the z coordinate in a z-buffer, and the w coordinate in a w-buffer—
16 at the current point is tested to see if it is smaller than the depth value stored in the
17 depth buffer. If the depth value of the polygon is smaller, it is stored in the depth
18 buffer and the color value from the polygon is written to the current point on the
19 rendering surface. If the depth value of the polygon at that point is larger, the next
20 polygon in the list is tested. This process is shown diagrammatically in Fig. 2.

21 There, notice that two polygons 200, 202 overlap along a ray that is
22 associated with a pixel 204 of interest. When the 3-D scene is rasterized, each
23 point on the rendering surface is tested. Here, the corresponding location in depth
24 buffer 206 corresponding to pixel 204 is set to the largest possible value for the
25 scene. The color value on the rendering surface for this location can be set to a

1 background color. Polygons 200 and 202 are tested to ascertain whether they
2 intersect with the current coordinate on the rendering surface. Since both
3 polygons intersect with the current coordinate on the rendering surface, the depth
4 value of polygon 200 is tested to see whether it is smaller than the value in the
5 depth buffer. Here, since the depth value of the polygon 200 for the associated
6 coordinate is smaller than the current depth value, the depth value for polygon 200
7 is written to the depth buffer and the color value for the polygon is written to the
8 current point in the rendering surface (also referred to as the color buffer). Next,
9 with the depth buffer holding the depth value for polygon 200, the depth value of
10 polygon 202 is tested against the current depth value in the depth buffer. Since the
11 depth value of polygon 202 is smaller than the depth value of polygon 200, the
12 depth value of polygon 202 is written to the depth buffer and the color value for
13 polygon 202 is written to the current point on the rendering surface.

14 In this manner, in the ultimately rendered image, overlapping portions of
15 polygon 202 will occlude underlying portions of polygon 200.

16 Now consider what has become a fundamental problem in 3-D graphics—
17 that which is referred to as *transparent depth sorting*.

18 To appreciate this problem, consider that there are typically two different
19 types of pixels—opaque pixels and transparent pixels. Opaque pixels are those
20 pixels that pass no light from behind. Transparent pixels are those pixels that do
21 pass some degree of light from behind.

22 Consider now Fig. 3 which shows a viewer looking at a scene through one
23 exemplary pixel 300 on a screen. When an object is rendered by a 3-D graphics
24 system, if the object is to appear as a realistic representation of what a viewer
25 would see in the real world, then this pixel should represent all of the light

1 contributions, reflected back towards the viewer, that lie along a ray R. In this
2 example, ray R intersects three different objects—an opaque mountain 302, a first
3 transparent object 304 and a second transparent object 306. The furthest pixel
4 away from the viewer is pixel 302a which lies on the mountain. Because this pixel
5 is opaque, no other objects that might be disposed behind the mountain will make
6 a contribution to the ultimately rendered object.

7 In the real world, transparent objects 304, 306 cause the light that is
8 reflected back towards the viewer to be affected in some way. That is, assume that
9 objects 304, 306 are glass or windows that have some type of coloration. The
10 effect of these windows is to slightly dim or otherwise attenuate the light that is
11 associated with pixel 302a. In the real world, the viewer's eye effectively sums all
12 of the light contributions to provide a realistic image of the distant mountain. In
13 the 3-D graphics world, this is not as easy.

14 Specifically, assume that pixel 302a has associated color values that
15 describe how that pixel is to be rendered without any transparency effects applied.
16 The influence of the right side of object 304 at 304a, and the left side of object 304
17 at 304b, will change the color values of the associated pixel. Similarly, the
18 influence of the right side of object 306 at 306a, and the left side at 306b, will
19 further change the color values of the associated pixel.

20 Thus, if one wishes to accurately render pixel 302a, one should necessarily
21 take into account the transparency effects of these transparent objects.

22 The traditional depth buffering techniques described above do nothing to
23 alleviate this problem. Specifically, the traditional depth buffering techniques
24 essentially locate the closest pixel (i.e. the pixel with the smallest z value) and then
25

1 write the pixel's color values to the color buffer. Thus, traditional depth buffering
2 techniques do not take into account this transparency issue.

3 There have been attempts in the past to solve this particular transparency
4 depth sorting issue. For example, one solution to this problem is to push the
5 problem onto the application programmer. For example, the application
6 programmer might resolve this issue by drawing all of the opaque objects first, and
7 then perform some type of inexpensive bounding box or bounding sphere
8 processing, and present the resulting data to a graphics engine in back-to-front
9 order. This can unnecessarily burden the application programmer.

10 Another general scheme to attempt to solve the transparency depth sorting
11 problem is known as the "A-buffer" approach. This approach creates a linked list
12 of all of the pieces of data as a frame is being drawn. For every pixel in the frame
13 buffer, there is a linked list of fragments. These fragments embody the
14 contributions of the various objects that are in the scene. The A-buffer approach is
15 a very general method that essentially collects all of the linked list data for each
16 pixel, and after all of linked list data is collected, resolves the back to front issues,
17 on a pixel by pixel basis, after the scene is completely drawn. In the context of a
18 resource-rich environment where time is not a factor, this approach is acceptable
19 as the software program simply operates on the data as the data is provided to it.

20 One problem with the A-buffer approach, however, is most easily
21 appreciated in environments that are not necessarily resource-rich, and where time
22 is, in fact, a factor, e.g. the gaming environment where it is desirable to render real
23 time 3-D graphics. With the A-buffer approach, the size of the linked lists and all
24 of the data in the linked lists can be quite large. Today, it is not economical to
25 have a frame buffer that is so large as to support the size of the linked list.

1 Additionally, this approach does not even take into account textures, geometries
2 and the like. While the results that are produced using the A-buffer approach are
3 acceptably nice, the costs associated with attaining such results are not appropriate
4 for the real time environment.

5 Accordingly, this invention arose out of concerns associated with providing
6 improved graphics systems and methods.

7 8 **SUMMARY**

9 Methods and systems for transparent depth sorting are described. In
10 accordance with one embodiment, multiple depth buffers are utilized to sort depth
11 data associated with multiple transparent pixels that overlie one another. The
12 sorting of the depth data enables identification of an individual transparent pixel
13 that lies closest to an associated opaque pixel. With the closest individual
14 transparent pixel being identified, the transparency effect of the identified pixel
15 relative to the associated opaque pixel is computed. If additional overlying
16 transparent pixels remain, a next closest transparent pixel relative to the opaque
17 pixel is identified and, for the next closest pixel, the transparency effect is
18 computed relative to the transparency effect that was just computed.

19 20 **BRIEF DESCRIPTION OF THE DRAWINGS**

21 Fig. 1 is a block diagram that illustrates one type of computer graphics
22 system.

23 Fig. 2 is a diagram that illustrates a depth sorting process that utilizes a
24 depth buffer.

1 Fig. 3 is a diagram that illustrates aspects of a computer graphics scenario
2 in which multiple transparent objects are employed.

3 Fig. 4 illustrates an exemplary system that can be utilized to implement the
4 various embodiments described herein.

5 Fig. 5 is a block diagram that illustrates a transparent depth sorting
6 component in accordance with one embodiment.

7 Fig. 6 is a flow diagram and graphic that illustrates steps in a method in
8 accordance with one embodiment.

9 10 **DETAILED DESCRIPTION**

11 **Overview**

12 Reference will now be made in detail to exemplary embodiments, examples
13 of which are illustrated in the accompanying drawings. The described
14 embodiments are not intended to limit application of the claimed subject matter to
15 only the specific embodiments described. On the contrary, the claimed subject
16 matter is intended to cover alternatives, modifications and equivalents, which may
17 be included within the spirit and scope of various features of the described
18 embodiments.

19 Furthermore, in the following detailed description, numerous specific
20 details are set forth in order to provide a thorough understanding of the described
21 embodiments. It is quite possible, however, for the various embodiments to be
22 practiced without these specific details, but with details that are different, but still
23 within the spirit of the claimed subject matter. In some instances, well-known
24 methods, procedures, components, and circuits that are ancillary to, but support
25

1 the claimed embodiments have not been described in detail so as not to
2 unnecessarily obscure aspects of the embodiments that are described.

3 Some portions of the detailed descriptions which follow are presented in
4 terms of procedures, logic blocks, processing, and other symbolic representations
5 of operations on data bits within a computer memory or cache. These descriptions
6 and representations are the means used by those skilled in the data processing arts
7 to most effectively convey the substance of their work to others skilled in the art.
8 In the present application, a procedure, logic block, process, or the like, is
9 conceived to be a self-consistent sequence of steps or instructions leading to a
10 desired result. The steps are those requiring physical manipulations of physical
11 quantities. Usually, although not necessarily, these quantities take the form of
12 electrical or magnetic signals capable of being stored, transferred, combined,
13 compared, and otherwise manipulated in a computer system. It has proven
14 convenient at times, principally for reasons of common usage, to refer to these
15 signals as transactions, bits, values, elements, symbols, characters, fragments,
16 pixels, pixel data, or the like.

17 In the discussion that follows, terms such as "processing," "operating,"
18 "calculating," "determining," "displaying," or the like, refer to actions and
19 processes of a computer system or similar electronic computing device. The
20 computer system or similar electronic computing device manipulates and
21 transforms data represented as physical (electronic) quantities within the computer
22 system memories, registers or other such information storage, transmission or
23 display devices.

24 The embodiments described below pertain to a graphics subsystem that is
25 programmed or programmable to operate upon pixel or texel data that is to be

1 ultimately rendered to some type of display device. This graphics subsystem can
2 comprise an entire graphics engine, including a transform engine for geometry
3 calculations, a raster component comprising one or more of texture components,
4 specular components, fog components, and alpha blending components, and any
5 other components that can process pixel or texel data. In some embodiments, the
6 graphics subsystem can be embodied in an integrated circuit component.

7 8 **Exemplary System**

9 Fig. 4 illustrates an exemplary system 400 that can be utilized to implement
10 one or more of the embodiments described below. This system is provided for
11 exemplary purposes only and is not intended to limit application of the claimed
12 subject matter.

13 System 400 exemplifies a computer-controlled, graphics system for
14 generating complex or three-dimensional images. Computer system 400
15 comprises a bus or other communication means 402 for communicating
16 information, and a processor 404 coupled with bus 402 for processing information.
17 Computer system 400 further comprises a random access memory (RAM) or other
18 dynamic storage device 406 (e.g. main memory) coupled to bus 402 for storing
19 information and instructions to be executed by processor 404. Main memory 406
20 also may be used for storing temporary variables or other intermediate information
21 during execution of instructions by processor 404. A data storage device 408 is
22 coupled to bus 402 and is used for storing information and instructions.
23 Furthermore, signal input/output (I/O) communication device 410 can be used to
24 couple computer system 400 onto, for example, a network.
25

1 Computer system 400 can also be coupled via bus 402 to an alphanumeric
2 input device 412, including alphanumeric and other keys, which is used for
3 communicating information and command selections to processor 404. Another
4 type of user input device is mouse 414 (or a like device such as a trackball or
5 cursor direction keys) which is used for communicating direction information and
6 command selections to processor 404, and for controlling cursor movement on a
7 display device 416. This input device typically has two degrees of freedom in two
8 axes, a first axis (e.g., x) and a second axis (e.g., y), which allows the device to
9 specify positions in a plane.

10 In accordance with the described embodiments, also coupled to bus 402 is a
11 graphics subsystem 418. Processor 404 provides graphics subsystem 418 with
12 graphics data such as drawing commands, coordinate vertex data, and other data
13 related to an object's geometric position, color, and surface parameters. In general,
14 graphics subsystem 418 processes the graphical data, converts the graphical data
15 into a screen coordinate system, generates pixel data (e.g., color, shading, texture)
16 based on the primitives (e.g., points, lines, polygons, and meshes), and performs
17 blending, anti-aliasing, and other functions. The resulting data are stored in a
18 frame buffer 420. A display subsystem (not specifically shown) reads the frame
19 buffer and displays the image on display device 416.

20 System 400 can be embodied as any type of computer system in which
21 graphics rendering is to take place. In some embodiments, system 400 can be
22 embodied as a game system in which 3D graphics rendering is to take place. As
23 will become apparent below, the inventive embodiments can provide more realistic
24 3D rendering in a real time manner that is particularly well suited for the dynamic
25 environments in which game system are typically employed.

Exemplary Embodiment Overview

In the embodiments described below, a solution to the transparent depth sorting problem is provided. In accordance with the described embodiments, multiple depth buffers are employed to support a sorting process that is directed to incorporating the transparency effects of any intervening transparent pixels on an associated opaque pixel that lies behind it.

In accordance with the described techniques, after all of the opaque pixels are rendered to, for example, a rendering surface, the sorting process looks to identify, for an associated opaque pixel, the closest transparent pixel that would overlie it or, from a viewer's perspective, be in front of it. When the sorting process finds the closest transparent pixel, the rasterization pipeline processes the pixel data associated with the opaque pixel to take into account the effect that the closest transparent pixel has on the opaque pixel. This typically results in the color values for the pixel of interest being modified. Once the color values are modified, the color values are written to the frame buffer for the pixel of interest. The sorting process then looks to identify the next closest transparent pixel and, if one is found, the rasterization pipeline processes the pixel data associated with the opaque pixel of interest to take into account the effect that the next closest transparent pixel has on the opaque pixel. Again, this typically results in the color values for the pixel of interest being modified. Once the color values are modified, the modified color values are again written to the frame buffer for the pixel of interest. This sorting process continues until, for a given opaque pixel, no additional overlying transparent pixels are found.

1 The result of this process is that the transparency effects of overlying pixels
2 are taken into account, in back-to-front order, so that underlying opaque pixels
3 have associated color values that accurately represent what the pixel would look
4 like.

6 Using Multiple Depth Buffers to Effect Transparent Depth Sorting

7 In the example about to be described, multiple depth buffers are utilized to
8 effect transparent depth sorting. In this specific example, the depth buffers
9 comprise z-buffers. It is to be appreciated that the techniques described herein can
10 be employed in connection with w-buffers without departing from the spirit and
11 scope of the claimed subject matter. For example, in one embodiment an inverse
12 w-buffer that is Huffman encoded for varying precision can be used.

13 In the example about to be described, two physical z-buffers are employed.
14 The hardware in the graphics subsystem understands two pointers and parameters
15 for the two physical z-buffers. In accordance with one embodiment, both of the z-
16 buffers are readable, while only one of the z-buffers is writable. The z-buffer that
17 is both readable and writable is referred to as the “destination buffer”, while the
18 readable only z-buffer is referred to as the “source buffer”. Additionally, in the
19 embodiment about to be described, support is provided for the graphics software
20 to be able to flip which buffer is considered the source buffer and which buffer is
21 considered the destination buffer. The ability to flip which buffers are considered
22 as the source and destination buffers effectively provides for a logical-to-physical
23 mapping of the source destination buffers to the actual physical buffers, as will be
24 appreciated by the skilled artisan. In one embodiment, flipping the buffers is
25

1 accomplished utilizing a multiplexer as input to the buffers. To flip the buffers,
2 the multiplexer select signal is flipped.

3 Fig. 5 shows a transparent depth sorting component 500 in accordance with
4 one embodiment. This component can reside at any suitable location within the
5 graphics processing pipeline. In one embodiment, component 500 can reside in
6 the graphics subsystem, such as subsystem 418 in Fig. 4. Additionally, while the
7 individual components of the transparent depth sorting component 500 are shown
8 to reside inside the component, such is not intended to constitute a physical
9 limitation on the arrangement and location of components. Rather, the transparent
10 depth sorting component can constitute a logical arrangement of components.
11 Accordingly, individual constituent parts of the transparent depth sorting
12 component 500 may not necessarily reside physically inside the component 500.

13 As illustrated, component 500 comprises or makes use of a first z buffer
14 502 (designated Z_0), a second z buffer 504 (designated Z_1), a z writeback counter
15 506 and comparison logic 508. A frame buffer 510 is provided and can be written
16 to as a result of the processing that is undertaken by transparent depth sorting
17 component 500, as will become apparent below.

18 Fig. 6 is a flow diagram that describes steps in a transparent depth sorting
19 method in accordance with one embodiment. The method can be implemented in
20 connection with any suitable hardware, software, firmware or combination thereof.
21 In one embodiment, the method can be implemented using a transparent depth
22 sorting component such as the one illustrated and described in Fig. 5. Alternately
23 or additionally, the method about to be described can be implemented using
24 various components within a graphics subsystem, such as the one shown and
25 described in Fig. 4.

1 In the method about to be described, two physical z-buffers are employed—
2 a first one designated z_0 and a second one designated z_1 . In the Fig. 6 flow
3 diagram, the contents in the individual buffers at various points in the described
4 process will be indicated, just to the right of the flow diagram, underneath
5 columns designated " z_0 " and " z_1 " which are grouped underneath encircled
6 numbers which indicate a particular pass in the process. So, for example, the first
7 pass in the process is designated by an encircled "1", the second pass in the
8 process is designated by an encircled "2" and so on. This is intended to help the
9 reader follow along the described process. Also notice just beneath the columns
10 designated " z_0 " and " z_1 " appears an example that shows, from a viewer's
11 perspective, four different pixels that lie along a particular ray. In this example,
12 pixel A is an opaque pixel, and pixels B, C, and D constitute transparent pixels (in
13 reverse order of their closeness to the viewer) which form the basis of the example
14 that is utilized to explain this embodiment. In this particular example, the depth
15 values of the pixels increase the further away from the viewer that they appear.
16 Thus, the depth value for pixel A is larger than the depth value for pixel B, and so
17 on.

18 In addition, recall that the notion of a "destination buffer" and a "source
19 buffer" was introduced above. The destination buffer is a buffer that is both
20 readable and writable, while the source buffer is a buffer that is only readable. In
21 the explanation that follows, the physical z buffers that are designated as the
22 destination and source buffers will change. To assist in keeping track of these
23 changes, a parenthetical "dest" and "src" will appear in the column associated with
24 the physical buffer that bears the associated designation.

1 Step 600 maps first z buffer z_0 as the destination z-buffer and step 602
2 renders all of the opaque objects. The mapping of the first z buffer operates to
3 designate the second z buffer z_1 as the source z buffer. Rendering the opaque
4 objects involves finding the opaque pixels that are the closest opaque pixels to the
5 viewer. Thus, rendering all of the opaque objects effectively obscures individual
6 pixels that lie behind the closest opaque pixel. When objects or pixels are
7 rendered, what is meant is that a series of primitives that represent the objects are
8 drawn. In some embodiments, this means that the whole series of triangles that
9 represent the objects are drawn.

10 When the opaque objects are rendered, individual color values for the
11 opaque objects are written to the frame buffer and their corresponding depth
12 values are written to the first z buffer z_0 . Thus, in this example, for the illustrated
13 ray, steps 600 and 602 result in the depth value for pixel A being written to the
14 first z buffer z_0 . At this point, the first z-buffer holds all of the depth data for all of
15 the opaque pixels associated with the rendering surface.

16 What the algorithm will now attempt to do is, for a ray that is cast for a
17 particular x, y coordinate for the frame buffer, find the transparent pixel
18 contribution that is as close as possible in depth to the opaque pixel.

19 Accordingly, step 604 maps second z buffer z_1 as the destination z buffer
20 which effectively flips the logical mapping of the buffers, and steps 606 and 608
21 respectively clear the destination z buffer and initialize the destination z buffer to a
22 predetermined value. In this example, the predetermined value comprises its
23 smallest value. The smallest value in this example is the value associated with the
24 closest depth to the viewer—which is typically referred to as the “hither plane”.
25

1 Step 610 clears a “z writeback counter” (e.g. component 506 in Fig. 5),
2 which is utilized to keep track of writebacks that occur to the destination z buffer.
3 Step 612 sets the z compare logic to not write to the frame buffer, but to write to
4 the destination z buffer if a new z value is greater than the value in the destination
5 z buffer (i.e. the hither plane value) and the new z value is less than the value in
6 the source buffer (i.e. the depth value of pixel A). This step is directed to
7 ascertaining the depth value of the transparent pixel that is closest to the opaque
8 pixel. In this particular example, the result of performing this step writes the depth
9 value associated with pixel B into the destination buffer which, in this example, is
10 the second z buffer z_1 .

11 Step 614 renders all of the transparent objects. In the first pass the z buffer
12 value is obtained. In the second pass (described below), the color value for the z
13 buffer level found in the first pass is obtained, and the frame buffer is written to.
14 This step involves drawing the series of primitives that are associated with the
15 transparent objects. Step 616 ascertains whether the z writeback counter is equal
16 to 0. If the z writeback counter is equal to 0, meaning that no z writebacks
17 occurred, then the method terminates or exits for a particular pixel of the rendering
18 surface. Effectively, exiting this process means that all of the transparency effects
19 for a particular pixel on the rendering surface have been computed, back-to-front.
20 In this particular example, since a z writeback occurred to account to the depth
21 value of pixel B, the process does not exit.

22 Step 618 maps z_0 as the destination z buffer which effectively flips the
23 logical mapping of the z buffers. Step 620 sets the z compare logic to write to the
24 frame buffer (i.e. rendering surface) and the z buffer if the new z value is equal to
25 the value in the source z buffer (i.e. the value associated with pixel B) and the new

1 z value is less than the value in the destination z buffer (i.e. the depth value of
2 pixel A). Here, since the new z value is the value associated with pixel B, both
3 conditions are true, the frame buffer and the z buffer are written to. This
4 effectively writes the value associated with pixel B into the destination z buffer.
5 Step 622 then renders all of the transparent objects.

6 Accordingly, in the first pass, the process has found the backmost
7 transparent pixel (i.e. pixel B) and has written color values associated with the
8 transparency effects of pixel B out to the frame buffer. This provides, in the first
9 pass, a transparency effect that incorporates the contributions from pixel A and B.
10 In the pass about to be described, the transparency effects of pixel next closest to
11 pixel B (i.e. pixel C) will be computed. Following that pass, the transparency of
12 the next closest pixel (i.e. pixel D) will be computed.

13 Accordingly, step 622 returns to step 604 and maps second z buffer z_1 as
14 the destination z buffer which effectively flips the logical mapping of the buffers,
15 and steps 606 and 608 respectively clear the destination z buffer and initializes the
16 destination z buffer to its smallest value—i.e. the hither value. Step 610 clears the
17 z writeback counter and step 612 sets the z compare logic to not write to the frame
18 buffer, but to write to the destination z buffer if a new z value is greater than the
19 value in the destination z buffer (i.e. the hither plane value) and the new z value is
20 less than the value in the source buffer (i.e. the depth value of pixel B). This step
21 is directed to ascertaining the closest transparent pixel to pixel B. In this particular
22 example, the result of performing this step writes the depth value associated with
23 pixel C into the destination buffer which, in this example, is the second z buffer z_1 .

24 Step 614 renders all of the transparent objects and step 616 ascertains
25 whether the z writeback counter is equal to 0. If the z writeback counter is equal

1 to 0, meaning that no z writebacks occurred, then the method exits for a particular
2 pixel of the rendering surface. Effectively, exiting this process means that all of
3 the transparency effects for a particular pixel on the rendering surface have been
4 computed, back-to-front. In this particular example, since a z writeback occurred
5 to account to the depth value of pixel C, the process does not exit.

6 Step 618 maps z_0 as the destination z buffer which effectively flips the
7 logical mapping of the z buffers. Step 620 sets the z compare logic to write to the
8 frame buffer (i.e. rendering surface) and the z buffer if the new z value is equal to
9 the value in the source z buffer (i.e. the value associated with pixel C) and the new
10 z value is less than the value in the destination z buffer (i.e. the depth value of
11 pixel B). Here, since the new z value is the value associated with pixel C, both
12 conditions are true, the frame buffer and the z buffer are written to. This
13 effectively writes the value associated with pixel C into the destination z buffer.
14 Step 622 then renders all of the transparent objects. At this point in the process,
15 the transparency effects due to pixel C have been accounted for in the rendering
16 surface.

17 The method then returns to step 604 and maps second z buffer z_1 as the
18 destination z buffer which effectively flips the logical mapping of the buffers, and
19 steps 606 and 608 respectively clear the destination z buffer and initializes the
20 destination z buffer to its smallest value—i.e. the hither value. Step 610 clears the
21 z writeback counter and step 612 sets the z compare logic to not write to the frame
22 buffer, but to write to the destination z buffer if a new z value is greater than the
23 value in the destination z buffer (i.e. the hither plane value) and the new z value is
24 less than the value in the source buffer (i.e. the depth value of pixel C). This step
25 is directed to ascertaining the next closest transparent pixel to pixel C that lies

1 between pixel C and the viewer. In this particular example, the result of
2 performing this step writes the depth value associated with pixel D into the
3 destination buffer which, in this example, is the second z buffer z_1 .

4 Step 614 renders all of the transparent objects and step 616 ascertains
5 whether the z writeback counter is equal to 0. If the z writeback counter is equal
6 to 0, meaning that no z writebacks occurred, then the method exits for a particular
7 pixel of the rendering surface. Effectively, exiting this process means that all of
8 the transparency effects for a particular pixel on the rendering surface have been
9 computed, back-to-front. In this particular example, since a z writeback occurred
10 to account to the depth value of pixel D, the process does not exit.

11 Step 618 maps z_0 as the destination z buffer which effectively flips the
12 logical mapping of the z buffers. Step 620 sets the z compare logic to write to the
13 frame buffer (i.e. rendering surface) and the z buffer if the new z value is equal to
14 the value in the source z buffer (i.e. the value associated with pixel D) and the new
15 z value is less than the value in the destination z buffer (i.e. the depth value of
16 pixel C). Here, since the new z value is the value associated with pixel D, both
17 conditions are true, the frame buffer and the z buffer are written to. This
18 effectively writes the value associated with pixel D into the destination z buffer.
19 Step 622 then renders all of the transparent objects. At this point in the process,
20 the transparency effects due to pixel D have been accounted for in the rendering
21 surface. The method then returns to step 604.

22 Without going through the flow diagram again, with the depth value for
23 pixel D residing in the source z buffer due to the buffer flipping of step 604, the
24 flow diagram will effectively, at step 616, exit the process as the z writeback
25 counter will equal 0.

1 Thus, what has occurred at this point is that the individual transparent
2 pixels have been depth sorted, back-to-front, and their individual effects on the
3 opaque pixel have been taken into account in a realistic manner. That is, the
4 process described above effectively sorts the z values of individual pixels to find
5 the depth of the transparent pixel that is furthest away from the viewer and closest
6 to an associated opaque pixel that the transparent pixel affects. The transparency
7 effects of this furthest most transparent pixel are computed and written to the
8 frame buffer. Then, the process repeats itself and looks for the transparent pixel
9 that is the next furthest most pixel away from the viewer and closest to the last
10 transparent pixel. If found, the transparency effects of the next furthest most
11 transparent pixel is computed and written to the frame buffer. This process
12 continues until there are no more transparent pixels along an associated ray. Thus,
13 the transparency effects of individual transparent pixels that lie along a particular
14 ray are taken into account in a back-to-front manner which provides realistic, real
15 time computer graphics rendering.

16 17 **Conclusion**

18 The methods and systems described above can provide solutions to the
19 transparent depth sorting process that can result in more realistic 3D graphics
20 rendering, particularly in scenarios in which real time rendering is appropriate.

21 Although the invention has been described in language specific to structural
22 features and/or methodological steps, it is to be understood that the invention
23 defined in the appended claims is not necessarily limited to the specific features or
24 steps described. Rather, the specific features and steps are disclosed as preferred
25 forms of implementing the claimed invention.